Αναφορά 2ης Εργαστηριακής Άσκησης στο μάθημα
Δομές Δεδομένων & Αρχείων
Καπελώνης Χαρίλαος, ΑΜ: 2020030058

Α) Υλοποίηση ζητούμενων δομών δεδομένων και αρχείων σε Java:

Λίγα λόγια για τον κώδικα:

Το πρόγραμμα υλοποιεί κάθε ζητούμενη δομή (Δυαδικό Δέντρο Έρευνας *και* Ουρά Προτεραιότητας) με χρήση array και με δυναμική παραχώρηση μνήμης. Οι αντίστοιχες κλάσεις βρίσκονται στα πακέτα BinarySearchTree και Heap.
Το πακέτο myPackage αφορά κλάσεις βοηθητικές ή/και κοινές για τα δύο άλλα πακέτα, συγκεκριμένα έχει τις κλάσεις Main, Tester, SearchResult, την GFG η οποία είναι μία κλάση που βοηθάει στην εκτύπωση δυαδικού δέντρου (τη χρησιμοποιήσαμε κατά τη δημιουργία του προγράμματος) η οποία βρέθηκε στο διαδίκτυο (βλ. Γ-Πηγές) και την FileController η οποία χρησιμοποιείται για το διάβασμα των *.bin* αρχείων που δόθηκαν.
Γενικά να αναφέρουμε ότι τα κλειδιά που εισάγονται σε οποιαδήποτε δομή πάντα ελέγχονται για το αν υπάρχουν ήδη και, σε περίπτωση που υπάρχουν, *δεν* εισάγονται.

1) Δυαδικό Δέντρο Έρευνας (ΔΔΕ):

(α) με χρήση πεδίου αριθμών (array) μεγέθους Ν:

Για τη συγκεκριμένη υλοποίηση χρειαστήκαμε έναν **3xN** πίνακα ακεραίων και δύο ακέραιους που λειτουργούν ως references: i) στη ρίζα (**root**) και ii) στο διαθέσιμο index για insertion (**avail**).
Ο αλγόριθμος για την εισαγωγή συνοπτικά είναι ότι χρησιμοποιούμε ένα while-loop (θα μπορούσε να γίνει και αναδρομικά) και, ξεκινώντας από τη ρίζα (**curr = root**), συγκρίνουμε το curr με το **key** που θέλουμε να εισάγουμε και πάμε αντίστοιχα αριστερά ή δεξιά στα παιδιά.
Αντίστοιχη διαδικασία και για την διαγραφή, απλά εδώ έχουμε ξεχωριστό κώδικα για την περίπτωση που είναι η ρίζα του δέντρου. Γενικά, η διαγραφή είναι απαιτητική και αρκετά περίπλοκη σε αυτή την υλοποίηση και αυτό συμβαίνει γιατί ο κάθε κόμβος δεν έχει reference του γονέα του. Έτσι, αν αναζητάμε ένα στοιχείο για διαγραφή πρέπει να σταματήσουμε την αναζήτηση στον γονέα, γιατί αυτός είναι που θα αναλάβει ως παιδιά τα παιδιά του στοιχείου για διαγραφή (περίπτωση 4 όπου ο κόμβος προς διαγραφή έχει 2 παιδιά).

(β) με χρήση δυναμικής παραχώρησης μνήμης:

Ο κώδικας βρέθηκε στο διαδίκτυο και τροποποιήθηκε από εμάς για τις ανάγκες της εργασίας.
Εδώ έχουμε δυναμική παραχώρηση μνήμης σε αντίθεση με την προηγούμενη υλοποίηση. Προτιμήθηκε αναδρομή στην εισαγωγή, διαγραφή, αλλά και αναζήτηση. Όπως θα δούμε και παρακάτω κερδίζουμε σε μνήμη αλλά χάνουμε σε ταχύτητα. Χρειαζόμαστε μόνο ένα reference στο **TreeNode** της ρίζας και κατά κύριο λόγο λειτουργούμε όμοια με το (α), με τη διαφορά ότι έχουμε αντικείμενα TreeNodes.

2) Ουρά προτεραιότητας (Heap):

(α) με χρήση πεδίου αριθμών:

Ο κώδικας βρέθηκε στις διαλέξεις του μαθήματος και τροποποιήθηκε από εμάς για τις ανάγκες της εργασίας.
Εδώ έχουμε πίνακα ακεραίων, αλλά μονοδιάστατο, που αν τον πάρουμε σειριακά και τοποθετήσουμε τα στοιχεία του σε ένα δυαδικό δέντρο σχηματίζεται ένα complete binary tree (max heap) που η ρίζα κάθε υποδέντρου είναι μεγαλύτερη απο τα παιδιά της. Χρησιμοποιούμε πράξεις μέσα στον πίνακα: για ένα στοιχείο στη θέση r έχουμε ότι ο γονέας βρίσκεται με (r-1)/2 (κ.ο.κ. για τα παιδιά). Για την εισαγωγή, τοποθετούμε το

στοιχείο στο τέλος του πίνακα και το ανταλλάσσουμε αναδρομικά με τον γονέα του, αν είναι μεγαλύτερο. Για τη διαγραφή, ανταλλάζουμε το στοιχείο με το τελευταίο του πίνακα και το αγνοούμε (με μια ανάθεση **n--**) και το στοιχείο που ήταν τελευταίο το κατεβάζουμε με συγκρίσεις στη σωστή του θέση.
Σημειώνεται ότι εμείς χρησιμοποιούμε μόνο τη **removeMax()** που είναι και η πιο χρήσιμη σε τέτοιες δομές.

(β) με χρήση δυναμικής παραχώρησης μνήμης:

Εδώ υλοποιήσαμε την heap με δυναμική παραχώρηση μνήμης. Τα στοιχεία του **δέντρου** που δημιουργείται σχηματίζουν επίσης και μία **διπλά συνδεδεμένη λίστα**. Μπορούμε να φανταστούμε ότι έχουμε την ίδια υλοποίηση με τον πίνακα της (α) με τις διαφορές ότι 1) αντί για πίνακα (στατική) έχουμε λίστα (δυναμική) και 2) αντί για πράξεις για να βρούμε γονέα και παιδιά, ο κάθε κόμβος (**HeapNode**) έχει reference σε γονέα, παιδιά, *next* και *previous*. Επίσης, έχουμε reference στο τελευταίο στοιχείο που εισήχθη (**last** – χρήσιμο για διαγραφές) και στον πατέρα ο οποίος θα δεχθεί με την **insert(int key)** το επόμενο στοιχείο (**parent**). Στα υπόλοιπα σημεία, ακολουθήσαμε την ίδια λογική με το (α).

Β) Απόδοση και σύγκριση μεθόδων/ σχολιασμός αποτελεσμάτων:

Τρέξαμε το **Tester.test** 200 φορές και πήραμε τον μέσο όρο των αποτελεσμάτων, έτσι ώστε να έχουμε μία πιο ακριβή εικόνα της απόδοσης κάθε μεθόδου:

Μέθοδος	**Μέσος αριθμός συγκρίσεων / εισαγωγή (10^6 εισαγωγές)**	**Συνολικός χρόνος για 10^6 εισαγωγές**	**Μέσος αριθμός συγκρίσεων / ανά διαγραφή (100 διαγραφές)**	**Συνολικός χρόνος για 100 διαγραφές**
ΔΔΕ με δυναμική παραχώρηση μνήμης	53	0.40183 sec	56	1.04251 E-4 sec
ΔΔΕ με array	52	0.29125 sec	28	1.39502E-4 sec

Μέθοδος	**Συνολικός χρόνος κατασκευής όταν τα κλειδιά δίνονται όλα μαζί**	**Συνολικός χρόνος κατασκευής όταν τα κλειδιά εισάγονται το ένα μετά το άλλο**	**Μέσος αριθμός συγκρίσεων / εισαγωγή (10^6 εισαγωγές)**	**Μέσος αριθμός συγκρίσεων / ανά διαγραφή (100 διαγραφές)**	**Συνολικός χρόνος για 100 διαγραφές**
Ουρά προτεραιότητας με array	0.036538 sec	0.013800 sec	7	40	3.45E-5 sec
Ουρά προτεραιότητας με Δυναμική Παραχώρησης	0.058331 sec	0.0425799 sec	7	117	11.0E-5 sec

Τα αποτελέσματα είναι **λογικά**. Περιμέναμε **ταχύτητα** στις υλοποιήσεις με πίνακα (στατική), ενώ θα χάναμε σε **μνήμη**, αφού θα χρειάζεται να δεσμεύουμε μνήμη που μπορεί να μην χρησιμοποιηθεί, σε αντίθεση με τη δυναμική παραχώρηση μνήμης. Συγκεκριμένα:

1. για τον πρώτο πίνακα οι συγκρίσεις στην εισαγωγή είναι σχεδόν ίδιες. Όπως περιμέναμε, ο χρόνος εισαγωγής είναι μικρότερος στην υλοποίηση με πίνακα, ενώ για τις διαγραφές έχουμε διπλάσιο αριθμό στη δυναμική υλοποίηση, με τον χρόνο να είναι μεγαλύτερος όμως στη στατική υλοποίηση. Αυτό αιτιολογείται στο ότι ο χρόνος δεν αυξάνεται μόνο με τις συγκρίσεις αλλά και για άλλους λόγους, όπως οι αναθέσεις.
2. για τον δεύτερο πίνακα οι χρόνοι για την κατασκευή είναι μικρότεροι στην υλοποίηση με πίνακα. Οι συγκρίσεις στην εισαγωγή είναι ίδιες, ενώ στις διαγραφές σχεδόν τριπλάσιες στην δυναμική υλοποίηση. Ο χρόνος είναι αντίστοιχος της σχέσης των συγκρίσεων για τις διαγραφές.

Οι μέθοδοι με πίνακα είναι πιο αποδοτικές. Μιλάμε για μία γρήγορη εποχή με ραγδαία ανάπτυξη, στην οποία η μνήμη είναι πιο “φτηνή”, με την ταχύτητα να έχει την μεγαλύτερη σημασία. Για το ΔΔΕ παρατηρείται ≈25% κέρδος σε χρόνο στις εισαγωγές (ζημιά στις διαγραφές). Το κέρδος για το Heap είναι ≈35% και ≈67% αντίστοιχα για τις κατασκευές, ενώ ≈69% για τις διαγραφές. Τα ίδια ισχύουν και για την αναζήτηση τυχαίου κλειδιού. Βέβαια, οφείλεται και στον τρόπο που δουλεύουν οι υπολογιστές (cache γρηγορότερη από την CPU, βλ. Γ-Πηγές).

Γ) Πηγές:

https://www.baeldung.com/java-binary-tree

https://www.omnicalculator.com/math/percentage-increase

https://stackoverflow.com/questions/49742388/why-binary-search-array-is-slightly-faster-than-binary-search-tree

https://www.geeksforgeeks.org/print-binary-tree-2-dimensions/